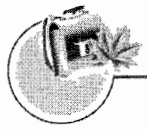
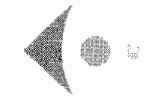


第6章

动态内存管理



程序要运行，就必须加载到内存中。早期的程序员，无论是否是嵌入式领域里的，都会想尽各种办法让自己写的程序尽可能少地占用内存。因为那时候内存非常得小，随便申请一块内存，可能就会超出系统最大内存值。也正是由于内存的限制，那时的程序功能并不强大。当然，随着技术的发展，内存越来越廉价，容量也越来越大。所以，现在的程序员似乎已经不大关心内存的占用问题了。

理想的内存模型具备两个条件：无限容量和无限速度。不幸的是，这两个条件在实际应用中都不可能满足。因此操作系统作为管理者，必须要肩负起管理内存的重任，让有限的内存发挥出无限的功能。

实际上，一个操作系统对内存的处理有两个基本原则。

一是怎样可以让内存的消耗最小。这一点的意义不言自明，内存消耗小则运行该操作系统的硬件成本就低，最终产品化后，企业就拥有了更低的价格优势。这里，我们绝对不是泛泛而谈，这种例子屡见不鲜。有一些芯片，如 AT91SAM7 系列，其内部集成了静态 RAM，这样，使用这种芯片的产品就可以省去外接RAM所带来的成本。如果一个操作系统精于内存节约之道，仅使用内置 RAM 就能够实现必要的功能，那么这样的产品无疑是具有市场价格竞争力的。

二是要尽可能地满足更多程序对内存的需求。节省归节省，但如果进程需要使用内存时，操作系统也不能吝惜。进程的运行是否足够自由是检验一个操作系统是否强大的重要指标。于是，在一个进程没有恶意使用内存的前

提下，尽最大努力满足进程对内存的需求就成为了操作系统处理内存的又一核心问题了。

上述两点只是原则上的，如果要在实际应用中去满足这两个原则，通常可以使用以下两种手段。

1）内存映射。将物理内存映射到虚拟内存中，变有限为无限。

2）有效管理。使用各种内存管理算法减少内存浪费，将一分钱掰成两半花。

对于第一种方法，我们在操作系统的启动一章中已有描述。理论上，经地址映射之后，用户程序可以使用的内存将会由原来的有限大小变成 4G 内存空间（对于 32 位 CPU 来说），这使得进程不会为系统可用内存的多少而纠缠不清，从而使进程能够去任意发挥。

然而，内存映射却不能解决所有的问题。这种方法只是看起来将内存扩大了，而实际的可用内存并没有丝毫增加。于是，如何对内存进行合理规划来减少内存的浪费，提高内存利用率，就成为了一个操作系统必须要考虑的问题。

当然，并不是所有的嵌入式操作系统都同时使用以上两种方法。很多嵌入式操作系统，比如 UCOS，都被设计成可以运行在一些没有内存管理单元的 CPU 中，所以不具备内存映射的能力。对于这些操作系统来说，使用适当的内存管理算法是解决内存问题的唯一途径。而对于那些高端嵌入式操作系统，在完成内存映射操作的基础之上也会双管齐下，对映射完成后的虚拟内存进行有效的管理。

于是，本章将会把重点放到内存管理的方法上，让我们自己的操作系统也拥有最适合进程生存的土壤。

6.1　伙伴算法

通常情况下，一个高级操作系统必须要给进程提供基本的、能够在任意时刻申请和释放任意大小内存的功能，就像 malloc 函数那样。然而，实现 malloc 函数并不简单。由于进程申请内存的大小是任意的，如果操作系统对

malloc 函数的实现方法不当，将直接导致一个不可避免的问题，那就是内存碎片。

通俗地讲，内存碎片就是内存被分割成很小很小的一些块，这些块虽然是空闲的，但是却小到无法使用。随着申请和释放次数的增加，内存将变得越来越不连续。最后，放眼望去，整个内存将只剩下碎片，没有有效的内存可用。所以减少内存浪费的核心是尽量避免产生内存碎片。

针对这样的问题，有很多种解决方法，伙伴算法就是其中之一。伙伴算法被证明是非常行之有效的一套内存管理算法，因此也被相当多的优秀操作系统所采用。本节中，我们就来介绍一下伙伴算法的原理，然后再结合实例深入理解一下操作系统中伙伴算法的具体实现。

什么是伙伴算法？简而言之，伙伴算法就是将内存分成若干小块，然后尽可能以最适合的方式满足程序内存需求的一种内存管理算法。伙伴算法的一大优势是它能完全避免外部碎片的产生。

那么什么又是外部碎片呢？我们来举个例子，假设系统可提供 1M 的内存空间供进程使用，由于算法设计等诸多原因，系统将这 1M 内存分成 4 个区域，让每个区域的默认大小为 300K。这样的结果是最后一个内存区域将不足 300K，此时，这个不足 300K 的内存区虽然理论上可以被使用，但由于容量偏小，将这段内存分配出去也许不能够满足程序的正常需要，同时操作系统也不得不专门去处理这段唯一的、不足默认 300K 大小的内存区域，这也为程序的设计带来了难度。这最后一段内存区域就像鸡肋一样，弃之不舍，食之无味，它就代表了我们所说的外部碎片。

当然，这里默认内存大小为 300K 的假设过于灵活了。很多关于操作系统原理的书中都会将这些概念与硬件分页机制相结合。这样一来，默认内存大小将会被限制为页的大小，而外部碎片就很难再被利用了。

不过，既然我们的目的不是对内存管理算法做理论性研究，很多概念上的东西也不需要解释得太精确。通过一个形象的例子来理解问题，反而是一件好事。与外部碎片相对应的另一个概念是内部碎片。内部碎片是指当某一个程序申请了一块内存，而此时系统中却没有大小恰好合适的内存提供给该程序，于是系统只能给该程序分配一块稍大一点的内存，而这块内存中会有一小部分根本不会被使用，这一部分内存就叫做内部碎片。

还拿前面的那个例子来说明问题。如果一个程序只想向系统申请 298K

的内存，但操作系统怎么会有恰好合适的内存分配给它呢？唯一的办法就是直接分配给这个进程一个 300K 的内存。这样一来，进程只利用了其中的 298K，而余下的 2K 就变成内部碎片浪费掉了。

伙伴算法虽然能够避免外部碎片的产生，但这恰恰是以产生内部碎片为代价的。

6.1.1　伙伴算法的原理

伙伴算法是如何工作的呢？

想要使用伙伴算法来进行内存管理。首先，系统需要将一块平坦的内存划分成若干块，可以用“buddy”这个词来描述。当然，这些内存块并不是任意大小的。根据伙伴算法的经典理论，这些内存块的大小必须是 2 的整数次方。例如，一块 1M 的内存可以按照 64K、128K、256K 和 512K 等的数值分成若干块。其实按照什么数值分块还是有一定原则的。通常我们选择比可用内存数小的 2 的整数次方的最大值。例如，对于一个 8M 的内存，最大的内存块可以是 4M，而余下的 4M 内存用来划分小于 4M 的块。这样可以最大限度地保证程序能够从系统中申请到尽可能多的内存。

确定了最大块的数目之后，我们还需要确定最小内存块的大小。通常最小内存块大小的取值应该适中。如果取值过大，比如 1M，当一个程序需要申请 100K 内存时，操作系统也只能将 1M 的内存分配给该程序，从而使余下的 924K 内存变成碎片浪费掉。但如果最小内存块数值过小，则会加重操作系统搜索内存块的负荷。

在内存块划分完成后，系统就可以对内存进行分配和释放了。在分配内存时，首先从空闲的内存中搜索比申请的内存大的最小的内存块。如果这样的内存块存在，则将这块内存标记为“已用”，同时将该内存分配给应用程序。如果这样的内存不存在，则操作系统将寻找更大块的空闲内存，然后将这块内存平分成两部分，一部分返回给程序使用，另一部分作为空闲的内存块等待下一次被分配。

当程序释放内存时，操作系统首先将该内存回收，然后检查与该内存相邻的内存是否是同样大小并且同样处于空闲的状态。如果是，则将这两块内存合并，然后程序递归进行同样的检查。

以上就是伙伴算法的实现过程。不过，单纯的文字描述真的是非常枯燥、难以理解。下面我们通过一个例子，来更深入地理解一下伙伴算法的真正内涵。

假设系统中有 1M 大小的内存需要动态管理，按照伙伴算法的要求，需要将这 1M 的内存进行划分。这里，我们将这 1M 的内存划分成 64K、64K、128K、256K 和 512K 共五个部分，如图 6-1（a）所示。

此时，如果有一个程序 A 想要申请一块 45K 大小的内存，则系统会将第一块 64K 内存块分配给该程序，如图 6-1（b）所示。

然后程序 B 向系统申请一块 68K 大小的内存，系统会将 128K 内存块分配给该程序，如图 6-1（c）所示。

接下来，程序 C 要申请一块大小为 35K 的内存，系统将空闲的 64K 内存分配给该程序，如图 6-1（d）所示。

之后程序 D 需要一块大小为 90K 的内存。当程序 D 提出申请时，系统本该分配给程序 D 一块 128K 大小的内存，但此时内存中已经没有空闲的 128K 内存块了，于是根据伙伴算法的原理，系统会将 256K 大小的内存块平分，将其中一块分配给程序 D，另一块作为一个空闲内存块保留，等待以后使用，如图 6-1（e）所示。

紧接着，程序 C 释放了它申请的 64K 内存。在内存释放的同时，系统还负责检查与之相邻并且同样大小的内存是否也空闲，由于此时程序 A 并没有释放它的内存，所以系统只会将程序 C 的 64K 内存回收，如图 6-1（f）所示。

然后程序 A 也释放掉由它申请的内存，系统随即发现与之相邻且大小相同的一段内存块恰好也处于空闲状态。于是，将二者合并，如图 6-1（g）那样。

之后程序 B 释放掉它的 128K 内存，系统将这块内存与相邻的 128K 内存合并成 256K 的空闲内存，如图 6-1（h）所示。

最后程序 D 也释放掉了它的内存，经过三次合并后，系统得到了一块 1024K 的完整内存，如图 6-1（i）所示。

结合上面的例子，原本枯燥无味的算法原理现在看起来是不是很简单了呢？下面我们趁热打铁，尝试在自己的操作系统中实现伙伴算法。

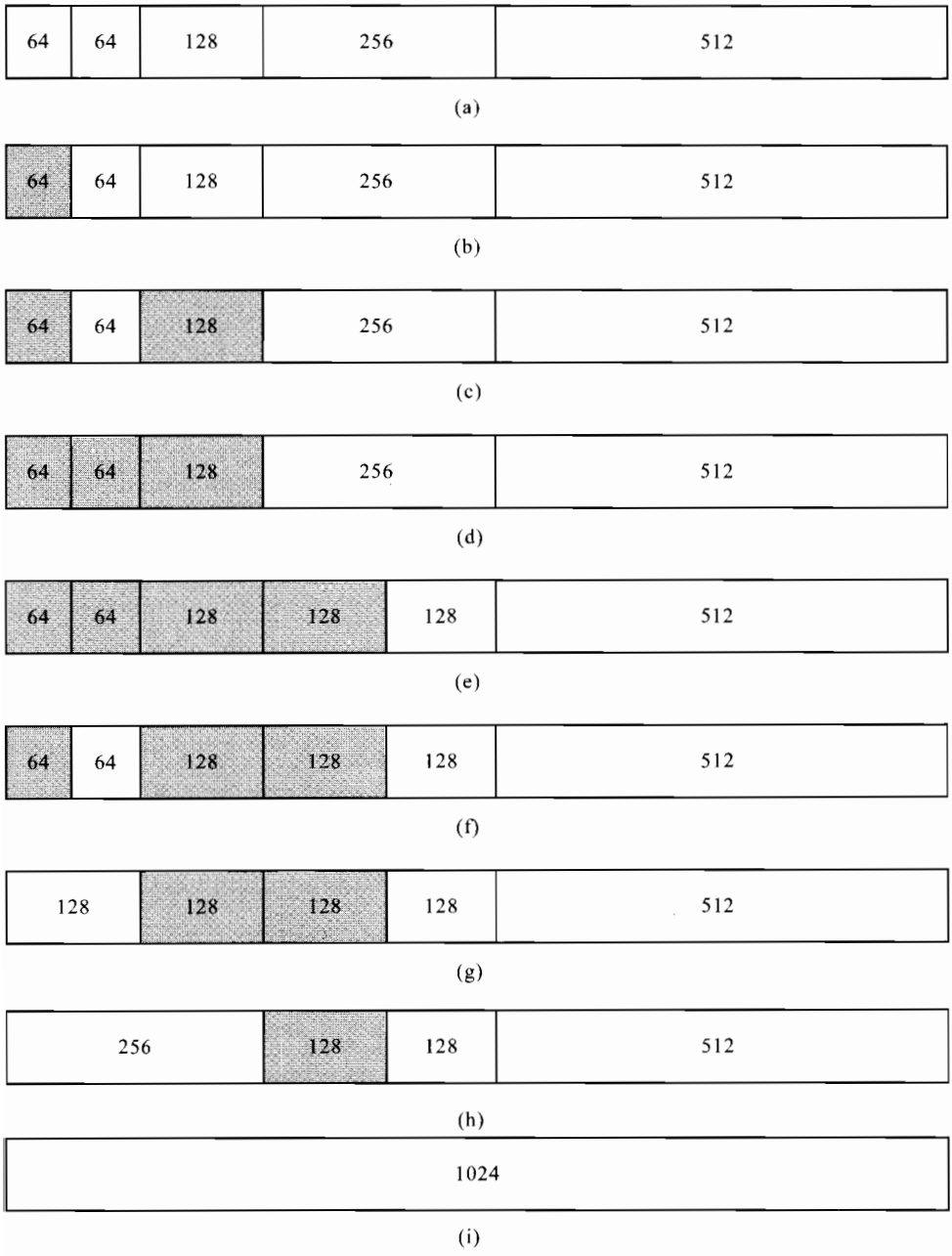

64
64
128
256
512
(a)
64
64
128
256
512
(b)
64
64
128
256
512
(c)
64
64
128
256
512
(d)
64
64
128
128
128
512
(e)
64
64
128
128
128
512
(f)
128
128
128
128
512
(g)
256
128
128
512
(h)
1024
(i)

图 6-1　伙伴算法示例

6.1.2 伙伴算法的实现

对于一个操作系统来说，所承担的内存管理角色较之于用户程序有着明显的不同。此时由于内存使用上受到了限制，操作系统还不具备动态数据管理的有效方法。于是，我们不得不只采用静态的办法来管理算法中需要的动态数据。在实现伙伴算法之前，对要管理的内存进行合理的规划是非常必要的。

6.1.2.1 内存规划

目前在我们的操作系统所使用的 8M 内存中，开头的部分是用来存储内核代码本身的，而结尾则用来存储各模式的运行堆栈、页表等系统关键数据。所以，能够进行动态管理的内存区域仅是中间的那一部分。

内存分区情况如图 6-2 所示。

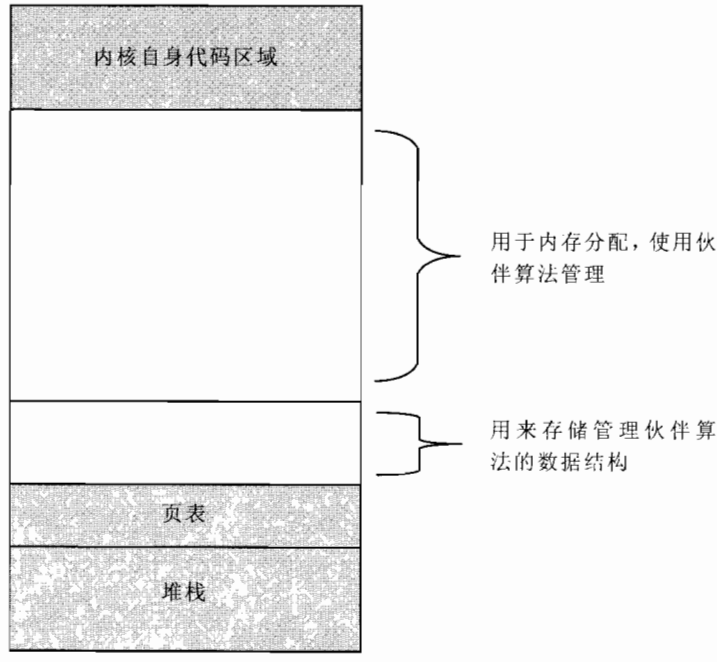


图 6-2 内存分区情况

在图 6-2 中，虽然整个可用内存中间的绝大部分都可以作为待分配的内

存进行管理，但同时，我们也必须留出一段特殊的内存空间用来存储管理伙伴算法所必需的数据结构。参考其他操作系统的命名方法，我们不如将用来管理伙伴算法的这一数据结构命名为 page。

代码 6-1

```
struct page {
    unsigned int vaddr;
    unsigned int flags;
    int order;
    struct kmem_cache *cachep;
    struct list_head list;
};
```

回忆一下我们对伙伴算法原理的描述。伙伴算法要求将内存分割成若干块，这些块最小不能低于一个数值，前面的例子中，这个值是 64K。任何一个大于这个值的内存块都必须是这一数值的 2 的 n 次方倍，也就是我们前面例子中的 128K、256K、512K，等等。

于是，我们将这个最小的内存区域叫做一个“页”，也就是前文所说的 64K 大小的内存区域。这里需要先强调一下，因为 page 翻译成中文也叫页，所以为了避免混淆，以后凡是提到 page 这个概念时，我们指的就是代码 6-1 这个数据结构，而当我们需要描述被分配的内存块时，我们就用中文“页”这个词来表达。

读者朋友可能会觉得把被分配的内存块叫做 block 或 area，似乎概念上会更清晰些。我们这样命名其实包含了一个非常重要的信息，有些朋友可能已经猜到了，没错，一个 page 是用来描述一个“页”的，反过来也是一样，一个“页”只能由一个 page 来描述，它们之间是一对一的映射关系。

既然一个 page 结构是用来描述一块最基本的内存区，该结构体中必然含有描述内存区域基本信息的成员，如起始地址、“页”的大小，等等。根据伙伴算法的描述，一个“页”的大小必然是一个固定的值。这里我们会选择与 MMU 的内存管理最小值相一致的值，在我们前面的例子中这个值是 4K。

但请注意，4K 这个值绝不是我们一拍脑门想出来的，里面也有些玄机。通过这种方法，系统可以在使用伙伴算法分配内存的同时将这一内存区映射到其他地址之中。当系统在执行用户应用程序时，这一点就显得相当必要了，

待分配的内存大小和待映射的内存大小相一致可以保证我们在为用户申请了一个“页”的内存空间的同时，以最小的代价实现对该页的映射，提高了系统的运行效率。

既然一个“页”的大小是固定不变的，那么就不需要在 struct page 结构体中体现出来了，我们可以非常简单地使用一个宏定义来描述“页”的大小。这样一来，在 struct page 结构体当中就仅需要一个描述内存区域起始位置的成员了。在代码 6-1 中，成员 vaddr 就扮演了这一重要角色；flags 成员则是作为一个标志，负责对 page 结构体和“页”进行控制；order 成员专门用于伙伴算法的管理，这个值可以是任意正数、0 或-1；成员 counter 是一个计数器，用来描述这块内存区被使用了多少次，如果 counter 值为零，代表没有任何进程使用该“页”，那么就可以将它当成空闲“页”处理；cachep 成员是一个 kmem_cache 类型结构体的指针，关于这个指针的内容，我们稍后会详细介绍。

最后一个成员是 struct list_head 结构体，该结构体是一个通用双向链表，其作用就是实现一个双向链表，将一系列同样大小的空闲内存连接起来。接下来我们会花些时间介绍一下这个通用链表结构的原理和使用方法。这么做的原因，一方面是由于该链表结构功能强大，被广泛地应用在各种程序之中，另一方面是因为它具有一定的抽象性，非常通用。掌握了它就能够在今后的编程过程中轻松运用，并发挥出巨大的作用了。

另外，我们这里的通用链表结构其实就来源于 Linux，因此，如果读者对 Linux 内核比较了解的话，也可以略过这部分内容。

6.1.2.2 通用链表结构

直接来看代码：

代码 6-2

```
struct list_head {
        struct list_head *next, *prev;
};

static inline void INIT_LIST_HEAD(struct list_head *list)
{
        list->next = list;
        list->prev = list;
```

```
}

static inline void __list_add(struct list_head *new_lst,
                struct list_head *prev,
                struct list_head *next)
{
        next->prev = new_lst;
        new_lst->next = next;
        new_lst->prev = prev;
        prev->next = new_lst;
}

static inline void list_add(struct list_head *new_lst, struct
list_head *head)
{
        __list_add(new_lst, head, head->next);
}

static inline void list_add_tail(struct list_head *new_lst,
struct list_head *head)
{
        __list_add(new_lst, head->prev, head);
}

static inline void __list_del(struct list_head * prev, struct
list_head * next)
{
        next->prev = prev;
        prev->next = next;
}

static inline void list_del(struct list_head * entry)
{
        __list_del(entry->prev,entry->next);
}

static inline void list_remove_chain(struct list_head
*ch,struct list_head *ct){
        ch->prev->next=ct->next;
        ct->next->prev=ch->prev;
}

static inline void list_add_chain(struct list_head *ch,struct
```

```
list_head *ct,struct list_head *head){
        ch->prev=head;
        ct->next=head->next;
        head->next->prev=ct;
        head->next=ch;
}

static inline void list_add_chain_tail(struct list_head
*ch,struct list_head *ct,struct list_head *head){
        ch->prev=head->prev;
        head->prev->next=ch;
        head->prev=ct;
        ct->next=head;
}

static inline int list_empty(const struct list_head *head)
{
        return head->next == head;
}

#define offsetof(TYPE, MEMBER) ((unsigned int) &((TYPE *)
0)->MEMBER)

#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \
        (type *)( (char *)__mptr - offsetof(type,member) );})

#define list_entry(ptr,type,member)     \
    container_of(ptr, type, member)

#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)
```

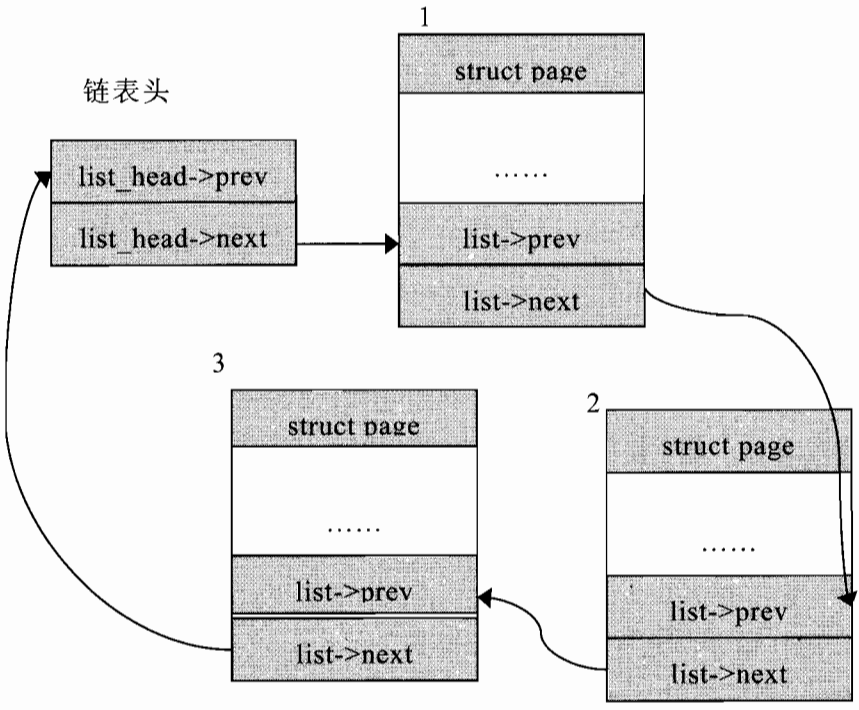
代码 6-2 中包含了对 struct list_head 结构的基本定义和操作方法。这其中，结构体的定义、初始化、添加、删除等内容其实都非常得常规。完全是按照双向链表的通用方法实现的，相信读者都看得明白。此时有的读者朋友可能会问，这样一个普普通通的链表结构又是怎样实现抽象化，发挥出通用性的优势的呢？这一切都源于一个构思精巧的宏——list_entry。

简单说来，如果一个结构体中包含一个 struct list_head 成员，那么 list_entry 这个宏能够从一个指向 list_head 成员的指针中计算出指向该结构

体的指针。

假设目前有三个 struct page 结构体，通过 list_head 连接在一起，如图 6-3 所示。

现在假设这三个 struct page 结构体所描述的“页”都为空闲页。那么，如果某个进程向操作系统申请使用一个页，就必然要从这三个“页”中分配。图 6-3 中的这三个 struct page 并不是通过某个类型为 struct page 的指针连接在一起的，而是使用了 struct list_head 结构体。



注：图中的 prev 指针示意省略未画

图 6-3　通用链表结构

内存分配的时候，系统首先会从链表头开始去查找它的 next 成员所指向的结构体。根据通用链表的定义，这个指针指向另一个 struct list_head 结构体。于是我们就得到了一个指向 struct list_head 结构体的指针，而这个指针恰好又是一个空闲的 struct page 结构体的成员了。

接下来又要如何通过 stuct list_head 指针得到 struct page 指针呢？这其

实很简单。看一下代码 6-1，struct page 一共包含 5 个成员，前 4 个成员都是整型，默认大小为 4 个字节。因此，在已知第 5 个成员（struct list_head）地址的情况下，想要求出结构体（struct page）首地址，只需要减去 sizeof（int）*4 个字节的偏移即可。

前面说的 list_entry 这个宏就是专门用来获取结构体首地址的，只不过它的实现方法更加“智能”，我们来具体分析一下。

首先来看看该宏的第一行表达式：

```
const typeof( ((type *)0)->member ) *__mptr = (ptr)
```

这里的 type 其实是宏的一个参数，使用时应该传递包含有 list_head 结构体的类型，ptr 是另外一个宏参数，代表指向 list_head 结构体的指针。member 也是 list_entry 的一个参数，它表示包含 list_head 结构体的 list_head 类型成员的名字。

根据前面的例子，struct page 中 list_head 类型成员名为 list。假设指针 struct list_head *p 恰好指向了 struct page 结构体的 list 成员，那么展开之后，上面的表达式就会像下面这样：

```
const typeof( ((struct page *)0)->list ) *__mptr = (p)
```

这样看来，这个表达式清晰很多，注意，其中的 typeof 是 GCC 针对 C 语言的一个扩展关键字，它的作用是得出其参数的具体类型。例如，有一个指针定义为 int foo，那么 typeof（foo）的最终结果就应该是 int。

将上述所有内容综合起来，整个表达式的含义就一目了然了。

宏 list_entry 首先将地址 0 强制转成 struct page 类型的指针，取其中的 list 成员，然后通过 typeof 提取它的类型，也就是 struct list_head。这就相当于定义了一个 const struct list_head 类型的指针数据__mptr，并让它的值等于已知的 struct list_head 指针 p。

然后，list_entry 宏又通过第二行的表达式返回一个指向 struct page 的指针，其形式如下：

```
(type *)( (char *)__mptr - offsetof(type,member) )
```

展开之后，该表达式等价于如下形式：

```
(struct page *)( (char *)__mptr - offsetof(struct page,list) )
```

宏 offsetof 能求出结构体首地址与它的某一个成员地址的偏移。其定义如下：

```
((unsigned int) &((TYPE *)0)->MEMBER)
```

展开之后，表达式变成：

```
((unsigned int) &((struct page *)0)->list)
```

这里的意思是，假设有一个 struct page 结构体从内存 0 位置开始，那么 list 成员的地址值就等于 list 成员地址与结构体首地址的偏移值。

宏 list_entry 最后返回的结果就是一个指向 struct page 结构体的指针。这样，针对 struct list_head 结构体的最重要的一种操作方法，我们就介绍完了。

如果有的读者此时正在思考下面这个问题的话，那您也一定闻到了这阵四溢的茶香了。

其实，ist_head 宏其实还蕴藏着一处妙笔， list_entry 这个宏完全可以写成如下形式：

```
(struct page *)((char *)(ptr)-offsetof(struct page,list))
```

乍看之下，这个表达式相比代码 6-2 中 list_entry 的定义更加简单，那为什么在代码 6-2 中，非要定义一个临时变量__mptr 并赋值为 ptr，然后代替 ptr 来计算地址值呢？

其实，这里使用 const typeof(((type *)0)->member) *__mptr 这种表达方法有一个好处，那就是可以强迫宏进行类型检查。

我们知道，宏定义本身并不会进行类型检查，而这种表达方法会迫使编译器检查 type 类型是否含有 member 成员。同时，如果程序在编译的时候打开了优化选项，那么__mptr 这个变量也会被优化掉，最终生成的代码与(struct page *)((char *)(ptr) - offsetof(struct_page,list))表达式生成的代码也没有什么不同。这样一来，使用代码 6-2 中 list_entry 的实现方法，既能够进行类型检查，提高了安全性，同时在效率上又没有丝毫的浪费。

正是由于有 list_entry 这个宏，当我们想将任何一种结构体类型连接成双向链表时，便不再需要专门给这个结构体写一套操作方法，而只需要将 struct list_head 作为这个结构体的成员，使用代码 6-2 中的一系列方法操作 struct list_head 链表。当需要得到结构体的指针时，只需要调用 list_entry 宏就可以了。

除了 list_entry 宏定义外，另一个比较常用的就是 list_for_each 宏。该宏能够遍历由 list_head 连接而成的整个链表。实际上，这个宏是通过一个 struct list_head 结构体的指针从链表头开始，每次都将当前操作链表的 next 值赋给该指针，直到该指针的值再次等于链表头的地址值为止，从而将整个链表循环了一遍。

6.1.2.3 与内存管理相关的宏定义

在掌握了一套构建链表的方法后，就可以开始进行伙伴算法的编码了。

首先我们需要参考图 6-2，规划一下内存边界。其原因正如前文所说，我们将内存按照“页”的大小进行划分的同时，还需要分配出同样多个用来存储 struct page 结构体的空间。

举例来说，假设一个“页”有 4K 大小，并且系统中共有 6M 空间可以利用，那么系统最多可以被划分出如下这么多个“页”。

6M/（4K+sizeof（struct page））

我们需要一些宏定义辅助我们在程序中实现内存划分。

代码 6-3

```
#define _MEM_END    0x30700000
#define _MEM_START  0x300f0000

#define PAGE_SHIFT   (12)
#define PAGE_SIZE    (1<<PAGE_SHIFT)
#define PAGE_MASK    (~(PAGE_SIZE-1))

#define KERNEL_MEM_END  (_MEM_END)

#define KERNEL_PAGING_START      \
((_MEM_START+(~PAGE_MASK)) & ((PAGE_MASK)))
#define  KERNEL_PAGING_END  (((KERNEL_MEM_END-\
                    KERNEL_PAGING_START)/(PAGE_SIZE+\
                    sizeof(struct page)))*(PAGE_SIZE)+\
                    KERNEL_PAGING_START)

#define  KERNEL_PAGE_NUM      ((KERNEL_PAGING_END-\
                    KERNEL_PAGING_START)/PAGE_SIZE)

#define  KERNEL_PAGE_END _MEM_END
```

```
#define  KERNEL_PAGE_START     (KERNEL_PAGE_END-\
                    KERNEL_PAGE_NUM*sizeof(struct page))
```

代码 6-3 通过一系列宏计算出了内存中各区域的大小。

_MEM_START 的值被定义为 0x3000f000，因为我们的虚拟硬件平台的默认物理内存地址是从 0x30000000 开始的，并且操作系统自身的代码也被加载到 0x30000000 处。那么，这里就简单地认为操作系统在运行时的大小将小于 0xf000。于是，从 0x3000f000 处开始向高地址延伸的内存就可以用来进行分配了。

同样地，_MEM_END 宏代表了可用内存的结束地址，即 0x30700000。在前面的例子中，我们都假设虚拟硬件平台共有 8M 物理内存。这其中，0x30700000~0x30800000 的内存区域被各模式的堆栈以及页表所占用，因此不能随意被修改。

这样一来，从_MEM_START 开始到_MEM_END 结束的内存空间，就都可以用来参与伙伴算法的分配了。如图 6-2 那样，这其中既包含了待分配的“页”，又包含了管理这些“页”的 struct page 结构体。

按照前文的描述，一个“页”的大小被设计成 4K，也就是代码 6-3 中 PAGE_SIZE 的值。而宏定义 KERNEL_PAGING_START 代表的则是_MEM_START 的值按照 PAGE_SIZE 对齐的地址。这里的对齐指的是能不能被整除。如果_MEM_START 本身就是按照“页”对齐的，那么这里的 KERNEL_PAGING_START 的值就应该等于_MEM_START，而如果_MEM_START 的值不是按页对齐的，如 0x3000f010 这样的值，那么 KERNEL_PAGING_START 的值就会向下取整，其结果最终会是 0x30010000 这个值。

KERNEL_PAGING_END 值的计算就稍微复杂了点。首先需要通过_MEM_END-KERNEL_PAGING_START 来计算可用内存的大小，然后再用这个值除以 PAGE_SIZE+sizeof(struct page)来计算该可用内存区一共能容纳多少个页，接着再用这个值乘以 PAGE_SIZE，之后再加上 KERNEL_PAGING_START，最终得到的就是内存中被分配的“页”的结束地址了。

知道了页的起始地址和结束地址之后，我们便可以通过下面这个表达式来计算系统中一共包含多少个“页”。

```
(KERNEL_PAGING_END-KERNEL_PAGING_START)/PAGE_SIZE
```

这个值被定义成了 KERNEL_PAGE_NUM 宏。当然，有多少个“页”，就会有多少个 struct page 结构体。于是，如果用于存放 struct page 结构体的内存区域到_MEM_END 结束的话，那么这一区域的起始地址就应该是：

```
(KERNEL_PAGE_END-KERNEL_PAGE_NUM*sizeof(struct page))
```

也就是代码中 KERNEL_PAGE_START 这个值。

这样，我们便完成了操作系统的内存划分。当所有准备工作都已完成后，接下来我们就一起来看看伙伴算法的核心实现。下面首先介绍伙伴算法是如何初始化的。

6.1.2.4 buddy 的初始化

初始化部分的代码如下：

代码 6-4

```
#define PAGE_AVAILABLE          0x00
#define PAGE_DIRTY              0x01
#define PAGE_PROTECT            0x02
#define PAGE_BUDDY_BUSY         0x04
#define PAGE_IN_CACHE           0x08

#define MAX_BUDDY_PAGE_NUM      (9)

#define  AVERAGE_PAGE_NUM_PER_BUDDY\
         (KERNEL_PAGE_NUM/MAX_BUDDY_PAGE_NUM)
#define  PAGE_NUM_FOR_MAX_BUDDY\
         ((1<<MAX_BUDDY_PAGE_NUM)-1)

struct list_head page_buddy[MAX_BUDDY_PAGE_NUM];

void init_page_buddy(void){
    int i;
    for(i=0;i<MAX_BUDDY_PAGE_NUM;i++){
        INIT_LIST_HEAD(&page_buddy[i]);
    }
}

void init_page_map(void){
    int i;
    struct page *pg=(struct page *)KERNEL_PAGE_START;
    init_page_buddy();
```

```
    for(i=0;i<(KERNEL_PAGE_NUM);pg++,i++){
        pg->vaddr=KERNEL_PAGING_START+i*PAGE_SIZE;
        pg->flags=PAGE_AVAILABLE;
        INIT_LIST_HEAD(&(pg->list));

        if(i<(KERNEL_PAGE_NUM&\
                (~PAGE_NUM_FOR_MAX_BUDDY))){
            if((i&PAGE_NUM_FOR_MAX_BUDDY)==0){
                pg->order=MAX_BUDDY_PAGE_NUM-1;
            }else{
                pg->order=-1;
            }
            list_add_tail(&(pg->list),\
                &page_buddy[MAX_BUDDY_PAGE_NUM-1]);
        }else{
            pg->order=0;
            list_add_tail(&(pg->list),&page_buddy[0]);
        }
    }
}
```

代码 6-4 似乎长了一些，我们慢慢来看。首先，程序定义了一个宏 MAX_BUDDY_PAGE_NUM，这个值与系统分配的最大内存的阶数有关。回忆一下前文中对伙伴算法原理的描述，在伙伴算法中 buddy 代表了一个可供分配的内存区。一个 buddy 的大小不是任意的，而必须是最小内存区大小的 2 的整数次方倍，如 4K、8K、16K、32K…。那么 MAX_BUDDY_PAGE_NUM 宏描述的就是所有不同大小的 buddy 的个数。

为了简化程序设计，此处我们选择了一个定值 9，表示从 2 的 0 次方一直到 2 的 8 次方共有 9 组 buddy 可以分配。换句话说，伙伴算法允许至少申请一个“页”，也就是 4K 大小的内存，同时也允许我们最大申请 256 个页，即 1M 内存。相对于系统不足 7M 的可用内存空间来说，这个范围还算合理。

参考伙伴算法的原理，为了能够更好地管理各组不同大小的 buddy，我们需要将同组的 buddy 连接成双向链表。结合通用链表一节中的描述，这需要系统中定义 MAX_BUDDY_PAGE_NUM 个链表头。

于是在代码 6-4 中，程序通过下面这行代码实现了链表头的定义：

```
struct list_head page_buddy[MAX_BUDDY_PAGE_NUM]
```

读者也可以通过索引数组成员来找到对应的链表头，而数组的索引值正是 buddy 的阶数，如 page_buddy[5]这个链表头所代表的双向链表都是大小为 2 的 5 次方的 buddy。

对这些链表头的初始化是通过 init_page_buddy 函数来实现的，该函数循环调用了 INIT_LIST_HEAD 函数，将每一个链表头的成员指针初始化为 NULL。INIT_LIST_HEAD 函数的实现方法见代码 6-2。

对各个 buddy 的初始化最终是由 init_page_map 函数实现的，这个函数首先调用 init_page_buddy 初始化链表头，然后通过循环分配 buddy，原则是尽可能分出空间最大的 buddy，当剩余空间不足以进行最大 buddy 的分配时，则该空间就按照最小的 buddy 进行分配。最终初始化的结果是系统中只包含有最大和最小的两种 buddy，如图 6-4 所示。

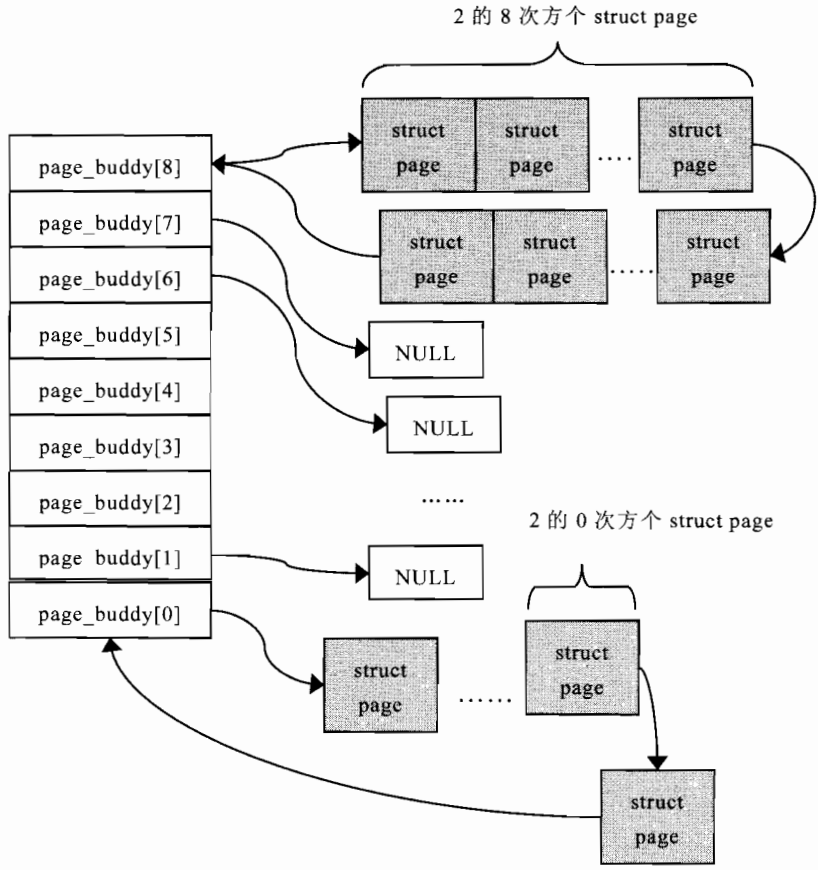


图 6-4　伙伴算法的初始化过程

读者朋友们此时也许正琢磨着，为什么要采用这样一种方式来初始化 buddy。根据伙伴算法，当系统中没有大小刚好合适的 buddy 时，系统可以将一个更高数量级的 buddy 拆成小块，再进行分配。但如果系统中初始化时就没有大块的 buddy，那么在首次分配时，有可能会出现没有可用 buddy 的情况。因此初始化 buddy 时，遵循大块优先的原则是比较合理的。

另一方面，统计结果说明系统中使用频率最高的内存绝大多数都小于 4K。因此，如果系统中不存在单“页”的 buddy 供程序频繁地申请，就必须从更高数量级的 buddy 中去分配。因此，预留一些单“页”的 buddy 供系统频繁使用也是提高效率的一种有效策略。

搞清楚了 buddy 的分配策略后，接下来我们来了解一下处理 struct page 结构的细节问题。对于任何一个 buddy 来说，其中都包含了一个或者多个“页”的空间，而每个页又都唯一对应着一个 struct page 结构体。于是，怎样来描述一个 buddy 就成了我们首先要考虑的问题。

这里我们使用的方法是，利用整个 buddy 中开始位置的 struct page 结构体来代表整个 buddy。于是，就需要一种手段区分出开头的 struct page 结构和普通的 struct page 结构，order 成员恰好可以解决这个问题。

在 buddy 初始化的过程中，我们首先要找到描述整个 buddy 最开头的那个“页”的 struct page 结构体，然后将这个结构体的 order 成员赋值成相应 buddy 的阶数，同时将那些普通的 struct page 结构体的 order 成员都置成-1，如图 6-5 所示。

这样一来，order 这个成员就具有了双重含义，一方面我们可以通过读取 order 的值直接得到当前 page 结构体属于哪个 buddy，另一方面，又可以利用它来判断这个 page 结构体是否是所属 buddy 的上边界。struct page 结构体的 order 成员在 buddy 的分配和释放过程中都会起到至关重要的作用。

在完成了系统中所有 struct page 结构体的初始化工作后，程序通过 list_add_tail 函数将所有 struct page 结构通过 list_head 结构体串联起来，添加到各自 buddy 链表头的尾部。

至此，伙伴算法初始化的部分就完成了。

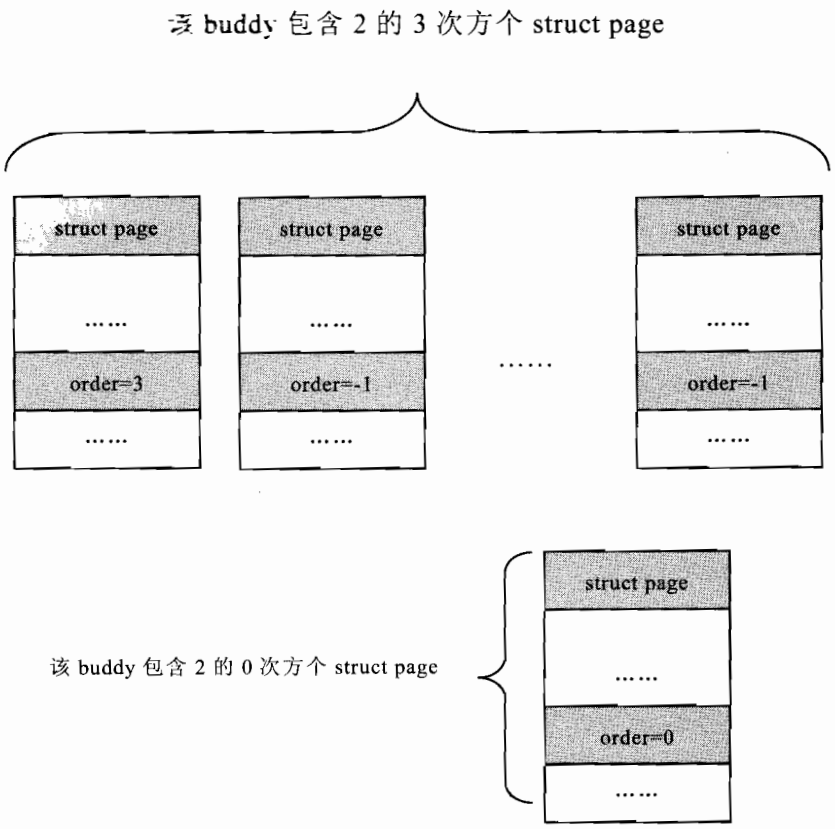


图 6-5　buddy 与 struct page 结构体的关系

6.1.2.5　buddy 的申请和释放

应用伙伴算法进行内存分配时，无非就是两种情况，其中一种情况是相对应的链表头下面恰好有空闲的 buddy，此时，我们只需要将一个 buddy 从链表中提出来即可，这种情况最为简单，通过下面几行代码就可以实现。

代码 6-5

```
……
#define BUDDY_END(x,order)   ((x)+(1<<(order))-1)
……
pg=list_entry(page_buddy[neworder].next,struct page,list);
tlst=&(BUDDY_END(pg,neworder)->list);
tlst->next->prev=&page_buddy[neworder];
```

```
page_buddy[neworder].next=tlst->next;
……
```

举例来说，如果我们要从系统中申请大小为 2 的 2 次方个“页”的 buddy，那么就需要首先找到阶数为 2 的链表头，该链表头的 next 成员所指向的地址其实就代表了一个空闲的 buddy，只不过这个地址的内容是一个 struct list_head 结构体。于是我们需要通过它得到 struct page 结构体的指针，而 list_entry 宏定义正好可以满足要求，这些都是我们前面介绍过的。

这个动作通过下面这行代码就可以实现：

```
pg=list_entry(page_buddy[order].next,struct page,list)
```

此时我们得到了内存中连续的 4 个“页”，但请注意，这里返回的并不是这块连续内存区的首地址，而是与这 4 个“页”中第一个“页”相对应的 struct page 结构体的地址。根据前面初始化部分的描述，这个 struct page 结构体的 order 成员的值应该是 2，而与另外 3 个“页”相对应的 struct page，它们的 order 成员都为-1。，虽然这个结构体能够代表整个 buddy，但想得到 buddy 首地址，还需要做一个小小的动作，即取出 struct page 结构体的 vaddr 成员。

在得到了连续的 buddy 之后，程序就必须把这个新的 buddy 从链表中拆下来，以免被再次分配。这就需要将相应 buddy 的链表头的 next 指针指向下一段空闲的 buddy。于是，我们需要得到标记下一段空闲 buddy 开始的 struct page 结构体取出它的 list 成员，赋值给 page_buddy[order]->next，等待下次分配。

申请 buddy 的示例如图 6-6 所示。

在代码 6-5 中，list_entry 函数通过 page_buddy[n]的 next 成员得到一个指向 struct page 的指针 pg，这里，n 就是该 buddy 的阶数。之后，宏 BUDDY_END 通过 pg 指针得到整个 buddy 结尾的那个 struct page。这样一来，buddy 所包含的起始和结束的 struct page 指针我们就都得到了，因此，可以将整个 buddy 从 page_buddy[n]链表中删除并返回给用户。此时，在 page_buddy[n]链表不为空的情况下，一个 buddy 就分配完成了。

buddy 的分配还有另外一种更复杂的情况，即当 page_buddy[n]链表为空时，就没有恰好空闲的 buddy 可用了。这样，根据伙伴算法的原理，程序只能去搜索 page_buddy[n+1]链表并对新的 buddy 进行拆分。

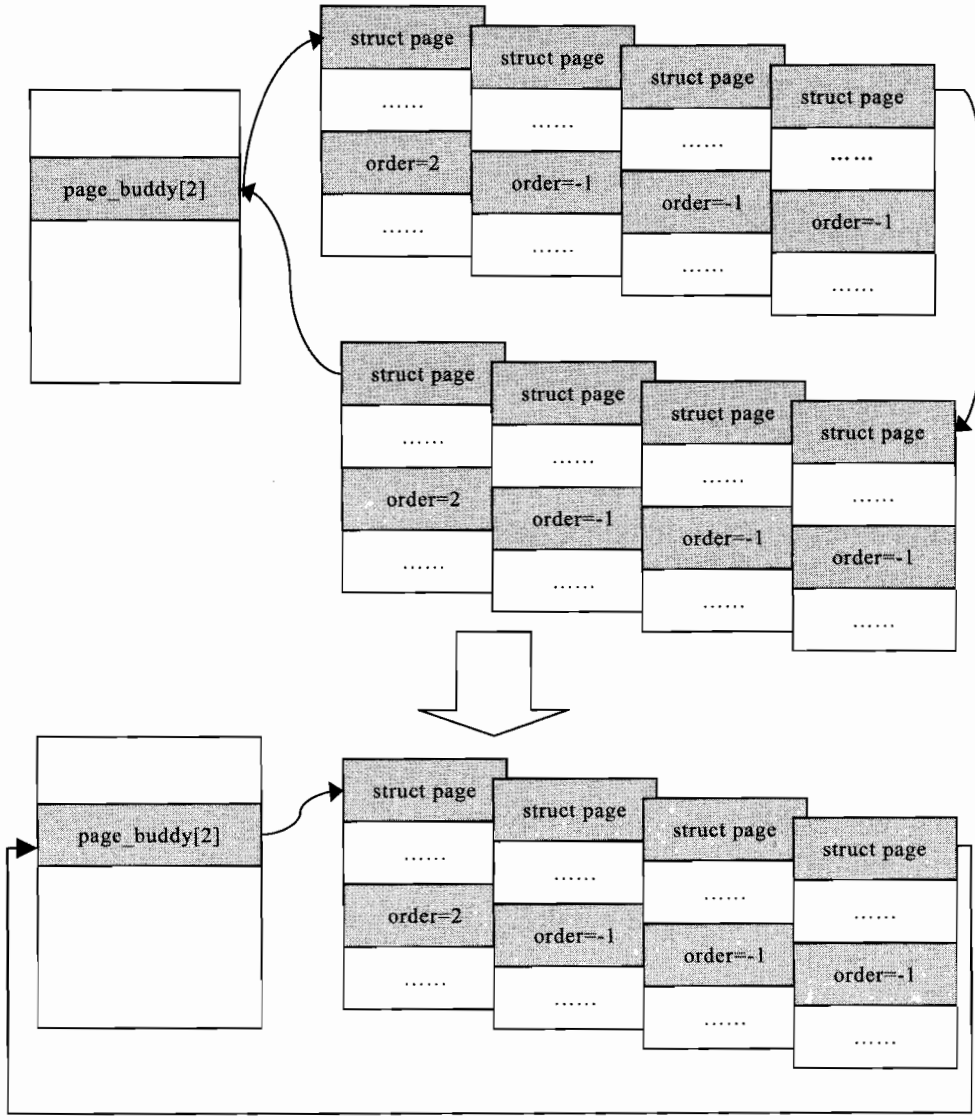


图 6-6　buddy 的申请

代码 6-6

```
#define PAGE_AVAILABLE        0x00
#define PAGE_DIRTY            0x01
#define PAGE_PROTECT          0x02
#define PAGE_BUDDY_BUSY       0x04
#define PAGE_IN_CACHE     0x08
```

```
#define BUDDY_END(x,order)          ((x)+(1<<(order))-1)
#define NEXT_BUDDY_START(x,order)   ((x)+(1<<(order)))
#define PREV_BUDDY_START(x,order)   ((x)-(1<<(order)))

struct page *get_pages_from_list(int order){
    unsigned int vaddr;
    int neworder=order;
    struct page *pg,*ret;
    struct list_head *tlst,*tlst1;
    for(;neworder<MAX_BUDDY_PAGE_NUM;neworder++){
        if(list_empty(&page_buddy[neworder])){
            continue;
        }else{
            pg=list_entry(page_buddy[neworder].next,struct
page,list);
            tlst=&(BUDDY_END(pg,neworder)->list);
            tlst->next->prev=&page_buddy[neworder];
            page_buddy[neworder].next=tlst->next;
            goto OUT_OK;
        }
    }
    return NULL;

OUT_OK:
    for(neworder--;neworder>=order;neworder--){
        tlst1=&(BUDDY_END(pg,neworder)->list);
        tlst=&(pg->list);
        pg=NEXT_BUDDY_START(pg,neworder);
        list_entry(tlst,struct page,list)->order=neworder;
        list_add_chain_tail(tlst,tlst1,&page_buddy[neworder]);
    }
    pg->flags|=PAGE_BUDDY_BUSY;
    pg->order=order;
    return pg;
}
```

代码 6-6 其实就是 buddy 分配函数的完整实现。在代码中，首先要经过从用户请求的 order 数到 MAX_BUDDY_PAGE_NUM 数的一个循环。在循环里，如果通过 list_empty 检查到 page_buddy[n]不为空，那就表示恰好有合适的 buddy 供分配，于是就可以采用与代码 6-5 一样的方法进行分配了。

但如果 page_buddy[n]为空，那就必须循环向上搜索，直到找到一个非空的 page_buddy 链表头为止，然后，还是使用相同的方法将整个 buddy 从 page_buddy[n]链表中拆下来。

这个时候，因为系统分配给用户的 buddy 阶数大于用户请求的 buddy 阶数。根据伙伴算法的描述，我们需要将多余的内存空间作为空闲的 buddy 挂到相应的 page_buddy 链表上。在代码 6-6 中，标签 OUT_OK 之后的 for 循环就实现了这个功能。

在这里，程序从实际分配给用户的 buddy 阶数减去 1 作为循环的开始，直到递减到用户请求的 buddy 阶数时就终止循环，在循环中实现 buddy 的拆分。例如，用户申请了一个阶数为 3 的 buddy，但系统只能分配一个阶数为 5 的 buddy。那么程序就需要从 4 开始循环，到 3 终止。这样，程序先拿出一个阶数为 4 的 buddy，再拿出一个阶数为 3 的 buddy，分别挂到相应的链表中。余下的阶数为 3 的 buddy 正好可以分配给用户。

在链表挂载的过程中，因为已知 buddy 的起始 struct page 结构，再通过 BUDDY_END 宏找到 buddy 中最后一个 struct page，就可以调用 list_add_chain_tail 把 buddy 挂到新的链表中去了。同时也要将新 buddy 的 order 数修改为相应的值。

最后，将得到的代表该 buddy 的 struct page 结构体中 flags 成员设置为 PAGE_BUDDY_BUSY，表示整个 buddy 正在被使用。

至此，伙伴算法中 buddy 的分配部分就完成了。

buddy 的释放正好与分配过程相反。这里我们也可以分两种情况来讨论。第一种情况是，与要释放的 buddy 前后相连的内存空间不能被合并，这种情况下，我们只需要在计算出 buddy 首尾的 struct page 结构体后，调用 list_add_chain 函数将其挂载到相应链表中即可。另外一种情况是，如果与要释放的 buddy 相连的内存空间可以合并，那就需要首先进行检查，然后再调用 list_add_chain 函数。其代码实现如下：

代码 6-7

```
void put_pages_to_list(struct page *pg,int order){
    struct page *tprev,*tnext;
    if(!(pg->flags&PAGE_BUDDY_BUSY)){
        return;
    }
```

```
    pg->flags&=~(PAGE_BUDDY_BUSY);
    for(;order<MAX_BUDDY_PAGE_NUM;order++){
        tnext=NEXT_BUDDY_START(pg,order);
        tprev=PREV_BUDDY_START(pg,order);
        if((!(tnext->flags&PAGE_BUDDY_BUSY))\
                &&(tnext->order==order)){
            pg->order++;
            tnext->order=-1;
            list_remove_chain(&(tnext->list),\
                    &(BUDDY_END(tnext,order)->list));
            BUDDY_END(pg,order)->list.next=&(tnext->list);
            tnext->list.prev=&(BUDDY_END(pg,order)->list);
            continue;
        }else if((!(tprev->flags&PAGE_BUDDY_BUSY))\
                    &&(tprev->order==order)){
            pg->order=-1;
            list_remove_chain(&(pg->list),\
                    &(BUDDY_END(pg,order)->list));
            BUDDY_END(tprev,order)->list.next=&(pg->list);
            pg->list.prev=&(BUDDY_END(tprev,order)->list);
            pg=tprev;
            pg->order++;
            continue;
        }else{
            break;
        }
    }

    list_add_chain(&(pg->list),&((tnext-1)->list),&page_buddy[order]);
}
```

程序首先通过 PREV_BUDDY_START 和 NEXT_BUDDY_START 这两个宏，得到与要释放的 buddy 相邻的前后两个 buddy 的起始 struct page 地址。然后判断这两个 struct page 是否设置了 PAGE_BUDDY_BUSY，同时还要判断 order 是否与要释放的 buddy 的 order 值相等。如果两个条件都具备，那么就调用 list_remove_chain，将这个 buddy 从原来的链表中拆下来，再修改链表指针将两个 buddy 连起来并更新 buddy 的 order 值。

put_pages_to_list 和 get_pages_to_list 两个函数都是底层函数，作为用户应用的直接函数接口，其实并不成熟。原因很简单，用户在申请内存时，希望得到的应该是内存首地址，而并不是所谓的 struct page 结构体。同时，当

一个 buddy 被成功申请下来，怎样去管理 buddy 中的每一个“页”，这两个函数都没能提供一个好的解决方法。

于是，针对上述问题，我们封装了这两个函数，提供给用户一组基本可用的内存分配功能。

代码 6-8

```
struct page *virt_to_page(unsigned int addr){
    unsigned int i;
    i=((addr)-KERNEL_PAGING_START)>>PAGE_SHIFT;
    if(i>KERNEL_PAGE_NUM)
        return NULL;
    return (struct page *)KERNEL_PAGE_START+i;
}

void *page_address(struct page *pg){
    return (void *)(pg->vaddr);
}

struct page *alloc_pages(unsigned int flag,int order){
    struct page *pg;
    int i;
    pg=get_pages_from_list(order);
    if(pg==NULL)
        return NULL;
    for(i=0;i<(1<<order);i++){
        (pg+i)->flags|=PAGE_DIRTY;
    }
    return pg;
}

void free_pages(struct page *pg,int order){
    int i;
    for(i=0;i<(1<<order);i++){
        (pg+i)->flags&=~PAGE_DIRTY;
    }
    put_pages_to_list(pg,order);
}

void *get_free_pages(unsigned int flag,int order){
    struct page * page;
    page = alloc_pages(flag, order);
```

```
    if (!page)
        return NULL;
    return page_address(page);
}

void put_free_pages(void *addr,int order){
    free_pages(virt_to_page((unsigned int)addr),order);
}
```

alloc_pages 首先调用 get_pages_from_list 进行“页”的分配，然后将分配得到的每一个 struct page 结构体的 flags 成员设置为 PAGE_DIRTY。这就表示程序可以利用该标志来判断每一个页的使用情况。

get_free_pages 函数在调用了 alloc_pages 函数之后，通过 page_address 将 struct page 结构体转换成已分配的内存块首地址，因此更有益于用户使用。

这里有一点要稍加说明，get_free_pages 和 alloc_pages 这两个函数都有一个 flag 参数。这个参数在这段例子代码中并没有什么作用，读者完全可以忽略它，但在实际应用中，flag 标志可以帮助我们对每一个要分配的“页”进行更高级的控制，如让某个“页”处在被保护的状态、禁止其他进程访问，等等。于是在代码 6-8 中，该标志便被保留了下来，以方便读者利用这段代码进行扩展。

与内存分配相对应，free_pages 和 put_free_pages 这两个函数分别在不同的层次中负责内存的释放工作。其中也调用了 virt_to_page 函数进行内存地址和 struct page 结构体之间的转换。具体的实现非常简单，我们就不再进行讨论了。

至此，在我们的操作系统中，一套基于伙伴算法的内存管理机制就正式建立起来了。让我们来运行这段代码，享受一下成功的喜悦吧！

6.1.2.6 算法的运行

要运行这段程序，首先需要在原来代码的基础上，将代码 6-8、6-7、6-6、6-4 的内容保存到名为“mem.c”的文件中。

然后修改 Makfile 中 OBJS 的内容为如下形式：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o
interrupt.o mem.o
```

接下来修改“boot.c”中的 plat_boot 函数，以便能调用内存分配的测试程序来检验一下伙伴算法的正确性。

代码 6-9

```
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    test_printk();
    //timer_init();

    init_page_map();
    char *p1,*p2,*p3,*p4;
    p1=(char *)get_free_pages(0,6);
    printk("the return address of get_free_pages %x\n",p1);
    p2=(char *)get_free_pages(0,6);
    printk("the return address of get_free_pages %x\n",p2);
    put_free_pages(p2,6);
    put_free_pages(p1,6);
    p3=(char *)get_free_pages(0,7);
    printk("the return address of get_free_pages %x\n",p3);
    p4=(char *)get_free_pages(0,7);
    printk("the return address of get_free_pages %x\n",p4);
    while(1);
}
```

为了使结果更加清楚，我们禁用了 timer_init 函数，以避免中断持续产生，影响输出结果。在 init_page_map 函数之后，程序调用了两个 get_free_pages 函数，分配两个 order 为 6 的 buddy，返回它们的内存首地址。然后，又连续调用两个 put_free_pages 函数将其释放。

此时，由于两个连续的 buddy 都处于空闲的状态，系统会将二者合并。紧接着，程序调用 put_free_pages 函数再次分配两个 order 为 7 的 buddy。打印出所有被分配内存的地址值供检验。

编译运行这些代码，运行结果如下：

```
arch: arm
```

```
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
testing printk
test string :::   this is %s test
test char ::: H
test digit ::: -256
test X ::: 0xffffff00
test unsigned ::: 4294967040
test zero ::: 0
the return address of get_free_pages 0x301b0000
the return address of get_free_pages 0x30170000
the return address of get_free_pages 0x300f0000
the return address of get_free_pages 0x30170000
```

从上面的结果中我们可以看出内存被反复申请和释放的一系列过程。相信读者此时已经领略到了伙伴算法的奥义了。

6.2　slab

优秀的内存管理方法对于一个操作系统来说是非常重要的。伙伴算法虽然解决了内存分配的外部碎片问题，但对于一个功能强大的操作系统来说，这还远远不够。正像我们前面提到过的，用户应用程序对内存的需求是频繁的和任意的。而伙伴算法作为一个基础内存管理算法，并不具备提供这种任意性的条件。

因此，我们还需要以伙伴算法为基础，实现另外的内存管理机制，为用户提供申请任意大小内存的可能。出于这样的目的，我们也有必要在这里重点介绍一下 slab 这个概念。

熟悉 Linux 内核的朋友一定会知道 slab。通俗地讲，slab 就是专门为某

一模块预先一次性申请一定数量的内存备用，当这个模块想要使用内存的时候，就不再需要从系统中分配内存了（因为从系统中申请内存的时间开销相对来说比较大），而是直接从预申请的内存中拿出一部分来使用，这样就提高了这个模块的内存申请速度。

6.2.1 使用 slab 的前提条件

这听起来似乎很不错，但 slab 也并不是万能的，要在合适的场合下使用它才能发挥作用。使用 slab 通常需要以下两个条件。

第一个条件是，当某一子系统需要频繁地申请和释放内存时，使用 slab 才会合理一些。如果某段程序申请和释放内存的频率不高，就没有必要预先申请一块很大的内存备用，然后再从这段私有空间中分配内存了。因为这样就意味着系统将会一次性损失过多内存，而由于内存请求的频率不高，也不会对系统性能有多大的提升。所以，对于频繁使用内存的程序来说，使用 slab 才有意义。

使用 slab 的另外一个条件是，利用 slab 申请的内存必须是大小固定的。只有固定内存大小才有可能实现内存的高速申请和释放。

使用 slab 的一个典型的例子是进程结构体。到目前为止，本书还没有涉及有关进程的问题。但可想而知，每一个进程都需要一个专门的结构体来描述它。每当一个新的进程产生的时候，系统都需要为该进程分配一段内存空间，用来存储该结构体。可想而知，这个结构体的分配具备了 slab 的两个条件，即第一，因为进程的产生和终止是一个频繁的过程，因此内存的申请和释放的频率就非常高，第二，因为结构体的大小是固定的，所以待分配的内存空间就固定了。因此，很多操作系统都将 slab 这种机制应用于进程结构体的分配中。

这里我们要强调一下，slab 这种机制并不是源于 Linux 的。Linux 所使用的 slab 分配器的基础是 Jeff Bonwick 为 SunOS 操作系统引入的一种算法。Jeff 的分配器是围绕对象缓存进行的。在内核中会为有限的对象分配大量内存，如文件描述符、进程结构体，等等。Jeff 发现对内核中普通对象进行初始化所需的时间超过了对其进行分配和释放所需的时间，因此他的结论是，不应该将内存释放回一个全局的内存池，而应保存在一个私有的内

存中。正是由于他的这种思想，最终产生了 slab 这种内存管理机制。在 Linux 中，除了 slab 之外，还有所谓的 slob 和 slub，这些内存分配机制名字不同，内部实现也大相径庭，不过它们的工作原理和根本目的都是一致的。这些不同的分配器各有优劣，分别适合于不同的应用场合。读者感兴趣的话，也可以查阅相关的文章，做进一步的了解。

既然对于一个操作系统来说，slab 机制如此地重要。那么接下来，不妨也在我们自己的操作系统中实现一个 slab。

6.2.2　slab 的组成

首先，我们要搞清楚 slab 的组成问题。从本质上讲，slab 也是一段内存空间，只不过这段内存空间只能包含那些阶数相同的 buddy，如图 6-7 所示。

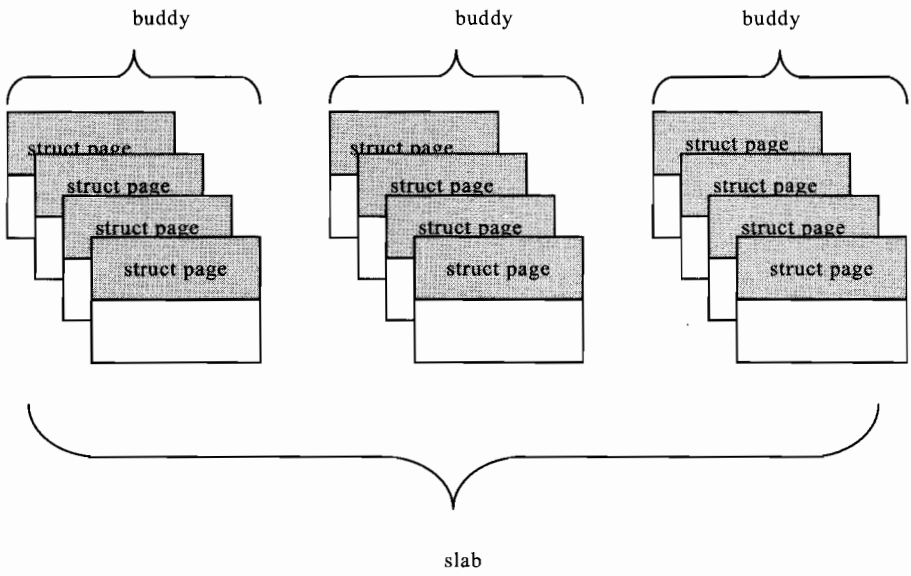


图 6-7　slab 的组成结构

下面的代码 6-10 定义了 kmem_cache 结构体，这是用来描述 slab 的一个基本结构体。此处我们取了一个与 Linux 内核相同的名字。当然，叫什么名字其实无关紧要，但这样做有一个好处，以后读者在接触 Linux 内核时，至少可以对其中的 kmem_cache 结构体有一个感性的认识。

代码 6-10

```
struct kmem_cache{
    unsigned int obj_size;
    unsigned int obj_nr;
    unsigned int page_order;
    unsigned int flags;
    struct page *head_page;
    struct page *end_page;
    void *nf_block;
};
```

在 kmem_cache 诸多成员中，obj_size 代表了该 slab 中子内存块的大小；obj_nr 表示 slab 中子内存块的数目；因为 slab 需要由一组 order 相同的 buddy 组成，那么这个 order 值就可以存储在 page_order 成员中；flags 成员用于对该 slab 进行控制，为了简化程序，这里我们同样省略掉对 flags 的操作；head_page 和 end_page 两个成员作为 struct page 类型的指针，指向了整个 slab 内存中开始和结束的那两个“页”所对应的 struct page 结构体；nf_block 成员最重要，它记录了该 slab 中下一个可用的子内存块的首地址。因此，当我们需要从 slab 中分配一个内存块时，直接返回 nf_block 即可。

也是因为同样的原因，我们也将采用与 Linux 相似的 slab 函数接口来操作 slab。

于是在使用 slab 进行内存分配时，就要首先调用 kmem_cache_create 函数来初始化一个 kmem_cache 结构体，然后再调用 kmem_cache_alloc 函数进行内存的分配，而释放已分配的内存则要依靠 kmem_cache_free 函数，当我们不再需要 slab 时，调用 kmem_cache_destroy 将这个 slab 内存区域彻底释放。于是，使用 slab 进行内存分配的步骤如图 6-8 所示。

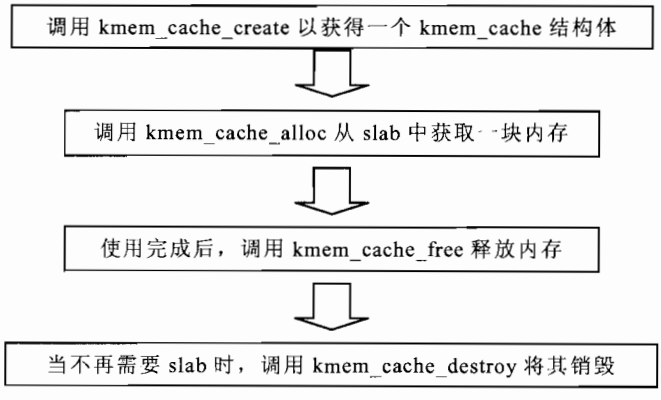


图 6-8　slab 使用流程

kmem_cache_create 函数能够初始化一个 struct kmem_cache 结构体。从代码 6-11 中我们可以看到函数 kmem_cache_create 共有三个参数，其中一个是 struct kmem_cache 结构体，另外两个是 slab 中子内存块的大小和控制标志。在该函数中，程序首先定义了一个指向 kmem_cache 中 nf_block 成员的指针 nf_blk，因为该成员也是指针，所以 nf_blk 其实是一个二级指针。然后，find_right_order 函数根据 slab 中子内存块的大小分析出究竟该 slab 需要阶数为几的 buddy，并把该阶数返回。

代码 6-11

```
#define KMEM_CACHE_DEFAULT_ORDER (0)
#define KMEM_CACHE_MAX_ORDER     (5)
#define KMEM_CACHE_SAVE_RATE     (0x5a)
#define KMEM_CACHE_PERCENT           (0x64)
#define KMEM_CACHE_MAX_WAST   (PAGE_SIZE-\
                        KMEM_CACHE_SAVE_RATE*\
                        PAGE_SIZE/KMEM_CACHE_PERCENT)

int find_right_order(unsigned int size){
    int order;
    for(order=0;order<=KMEM_CACHE_MAX_ORDER;order++){
        if(size<=(KMEM_CACHE_MAX_WAST)*(1<<order)){
            return order;
        }
    }
    if(size>(1<<order))
        return -1;
    return order;
}

int kmem_cache_line_object(void *head,unsigned int size,int
order){
    void **pl;
    char *p;
    pl=(void **)head;
    p=(char *)head+size;
    int i,s=PAGE_SIZE*(1<<order);
    for(i=0;s>size;i++,s-=size){
        *pl=(void *)p;
        pl=(void **)p;
        p=p+size;
```

```
    }
    if(s==size)
        i++;
    return i;
}

struct kmem_cache *kmem_cache_create(struct kmem_cache *cache,\
unsigned int size,unsigned int flags){
    void **nf_blk=&(cache->nf_block);
    int order=find_right_order(size);
    if(order==-1)
        return NULL;
    if((cache->head_page=alloc_pages(0,order))==NULL)
        return NULL;
    *nf_blk=page_address(cache->head_page);

    cache->obj_nr=kmem_cache_line_object(*nf_blk,size,order);
    cache->obj_size=size;
    cache->page_order=order;
    cache->flags=flags;
    cache->end_page=BUDDY_END(cache->head_page,order);
    cache->end_page->list.next=NULL;

    return cache;
}
```

find_right_order 函数就是在从 0 开始到 KMEM_CACHE_MAX_ORDER 的范围内查找当 order 等于多少的时候，能够容纳指定个数的子内存块。这里使用的是依照百分比的计算方法查看一个 size 是否占整个 buddy 的百分比小于某个特定值，然后把那个值返回。找到了正确的 order 值，就可以调用 alloc_pages 分配内存了。alloc_pages 函数返回的是 struct page 结构体的指针。于是我们将该指针赋值给 cache->head_page，为以后对 slab 的控制打下基础。同时，通过 page_address 获取 slab 内存首地址的指针赋值给 nf_blk 指针指向的内容，简单地说，就是赋值给了 cache->nf_block 这个变量。在一切准备就绪后，程序调用了 kmem_cache_line_object 初始化已分配的内存区域，然后在对 cache 的其他成员进行必要的初始化之后将指向 cache 的指针返回。

函数 kmem_cache_line_object 的主要目的就是将已分配好的 slab 内存连

接成如图 6-9 所示的样子。将 slab 内存分配成若干子块，每一个子块的大小由 kmem_cache 结构体的 obj_size 成员决定。子块分配完成后，再让每一个子块最开始的内存中存储下一个子块的起始地址。最后一个子块则存储 NULL 值，表示 slab 中已经没有子块可以提供使用。

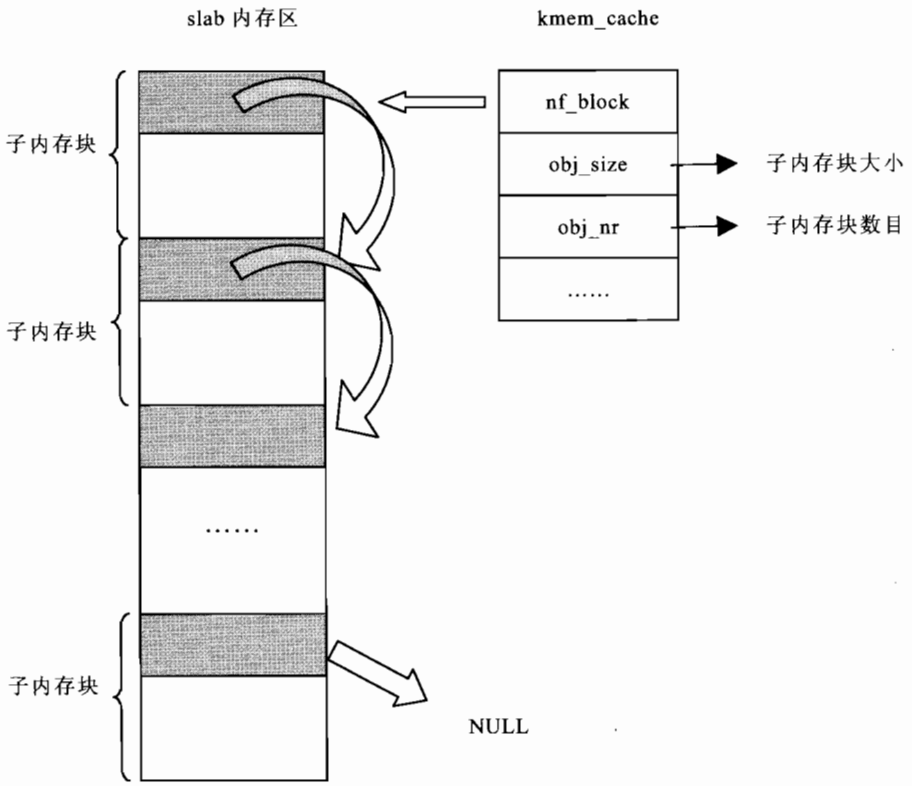


图 6-9　slab 的初始化

在初始化的时候，因为整个 slab 内存区都是空闲的，因此我们可以让 slab 中前后子块之间连续分布，一旦 slab 空间参与了分配，那么各子块之间就不会连续了，不过连续与否并不影响 slab 的正常使用。当然，一个 slab 内存区域可能并不能容纳整数倍的子内存块，余下最后一点内存区域，因为不足整个子块大小，就被浪费掉了。

6.2.3 通过 slab 进行内存分配

一旦 slab 内存区域初始化完成，我们便可以调用 slab 的分配函数从 slab 中获取内存了。

代码 6-12

```
void *kmem_cache_alloc(struct kmem_cache *cache,unsigned int flag){
    void *p;
    struct page *pg;
    if(cache==NULL)
        return NULL;
    void **nf_block=&(cache->nf_block);
    unsigned int *nr=&(cache->obj_nr);
    int order=cache->page_order;

    if(!*nr){
        if((pg=alloc_pages(0,order))==NULL)
            return NULL;
        *nf_block=page_address(pg);
        cache->end_page->list.next=&pg->list;
        cache->end_page=BUDDY_END(pg,order);
        cache->end_page->list.next=NULL;
        *nr+=kmem_cache_line_object(*nf_block,cache->obj_size,order);
    }

    (*nr)--;
    p=*nf_block;
    *nf_block=*(void **)p;
    pg=virt_to_page((unsigned int)p);
    pg->cachep=cache;
    return p;
}
```

函数 kmem_cache_alloc 能够从 slab 内存区中分配一个子内存块，并将这个子内存块的首地址返回。这里，我们也需要分两种情况来讨论。

第一种情况是，当 slab 中含有可分配的子块。此时，内存的分配相当简单，只需要将 kmem_cache 结构体的 nf_block 成员指向下一个空闲子内存块即可，这可以通过读取 nf_block 的内容来实现，将 obj_nr 成员的值自减 1，

然后将原来的 nf_block 值返回。

另一种情况略复杂一些。当经过了一段时间的分配和释放后，kmem_cache 的 obj_nr 成员为 0，slab 中已经没有可供分配的子内存块了。此时我们必须重新调用 alloc_page 函数分配一组新 buddy，同时调整 kmem_cache 相关成员，如 end_page 和 obj_nr 等。

kmem_cache_alloc 函数完整的实现方法就如代码 6-12 那样，需要注意的是函数结尾的部分有一行赋值语句：

```
pg->cachep=cache
```

这为 slab 子内存块的释放和进一步处理打下了基础。因为这样一来就可以确保每个 struct page 结构只属于一个 kmem_cache，同时，我们既可以通过虚拟地址找到所属的 struct page 结构，又可以通过 struct page 结构找到对应的 kmem_cache 结构体，从而保证了各个内存管理结构之间的相互转化。

6.2.4　内存空间的释放

相对于 slab 分配函数而言，slab 子内存块的释放函数就相对简单了。在这里，为使程序更加简化，我们并不承担内存回收的重任。也就是说，当内存不足的时候，我们不负责扫描空闲的内存块并将其释放，而是简简单单地返回 NULL。这样，无论 slab 占用多少内存，释放 slab 内存区这个动作只会发生在 slab 被销毁的时候。因此，对 slab 子内存块的释放不需要几行代码就可以实现了。

代码 6-13

```
void kmem_cache_free(struct kmem_cache *cache,void *objp){
    *(void **)objp=cache->nf_block;
    cache->nf_block=objp;
    cache->obj_nr++;
}
```

释放 slab 子内存块，无非就是更新 kmem_cache 结构体的 nf_block 成员，让它指向新被释放的内存地址，同时更新该地址的内容，指向原有的第一个子内存块首地址，这样，下次内存申请的时候，就可以将这个地址分配出去了。

6.2.5 slab 的销毁

最后，当这个程序不再需要 slab 进行内存分配时，就可以调用 kmem_cache_destroy 函数将这个内存空间销毁。而这里所谓的销毁，无非就是将组成 slab 内存区的各个小 buddy 依次释放掉而已，实现过程如下：

代码 6-14

```
void kmem_cache_destroy(struct kmem_cache *cache){
    int order=cache->page_order;
    struct page *pg=cache->head_page;
    struct list_head *list;
    while(1){
        list=BUDDY_END(pg,order)->list.next;
        free_pages(pg,order);
        if(list){
            pg=list_entry(list,struct page,list);
        }else{
            return;
        }
    }
}
```

看一下代码 6-14，在得到一个指向 struct page 结构体的指针和所属 buddy 的 order 值之后，就可以调用 free_pages 函数将这一个 buddy 释放。当然，此时函数还不能退出，因为一个 slab 内存区可能包含了多个相同阶数的 buddy。所以，程序需要通过循环来依次释放这些 buddy 直到结束。

至此，我们也在自己的操作系统中实现了一个简单的 slab 管理层。从结构上看，slab 的这种实现是能够处理内存高速请求的问题的。但这并不表示这段代码能够应付所有操作系统可能遇到的极端环境。

对于一些简单的操作系统来说，这段代码已经足够使用了。但如果我们的目的是设计出一套精简高效的通用操作系统，那么一些极端的情况也是必须要考虑进去的，其中之一就是内存的耗尽问题。一些典型的嵌入式操作系统，如 UCOS 等，因为应用程序和操作系统内核是一体的，所以程序和系统之间彼此完全了解。在这样的系统中，应用程序会充分了解系统内存的大小和使用情况，从而在设计时就避免了内存的过渡使用。然而，对于功能强

大的通用操作系统来说，应用程序和操作系统内核之间是完全分离的。应用程序并不充分了解系统的状况，也不太可能保证不去过度消耗内存。这样一来，系统内存在耗尽时要如何应对就成了通用操作系统必须要考虑的问题。

想要解决内存耗尽问题其实也不是难事。当系统中可用内存耗尽时，只需要去扫描已用的内存，看看是否有一些内存虽然被申请了但却没有被使用。

毫无疑问，slab 应该是首当其冲的。对本章内容理解相对透彻的朋友应该清楚，slab 就是预先申请一大块内存，然后留给某段私有程序慢慢用。因此，slab 中必然包含了许多空闲内存，这些内存不能被其他应用程序使用，同时也暂时没有被 slab 的拥有者所利用。

因此，当操作系统发现系统剩余内存不足以分配给某段程序时，就应该要求各个 slab“提前还款”了。不过，由于这部分内容已经违背了本书的编写目的，因此我们就不再详细讲述了。毕竟即使没有内存回收策略，我们的操作系统结构仍然是完整了，应用程序依然可以正常运行的。

slab 究竟有多重要，想一想一个操作系统中，究竟有多少类似于文件描述符这样的结构，就应该很清楚了。slab 仍然有它自己的适用范围，毕竟我们不能要求程序去适应操作系统，而应该想尽一切办法让操作系统尽可能满足程序的所有要求。

然而，读者朋友们也许在读完本节之后仍然没有掌握解决任意内存申请的方法。别急，slab 正是实现这种方法的一个基本手段。接下来的一节，我们将以此为基础，在操作系统中实现一个能够申请任意内存的函数。借鉴 Linux 的命名规则，这里我们也将这个函数命名为 kmalloc。

6.3　kmalloc 函数

kmalloc 函数应该怎样实现呢？

试想，我们预先定义若干个 kmem_cache 结构体，用来代表不同大小子内存块的 slab。这样，当某一个应用程序想要申请一段内存空间时，只需要就近找到一个合适的 slab 并从中分配一块内存即可。

例如，我们预先申请一组子内存块大小为 32 字节~128K 的 slab，每个 slab 之间能够管理的子内存块大小间隔 32 字节。当某个程序想申请 78 字节的内存，就可以从子内存块为 96B 的 slab 中拿出一块内存分配给它，其他大小亦是如此。这样，理论上每一次申请所浪费的内存数最多是 32 字节，通常这是可以接受的。如果想减少这种内存的浪费，可以将不同内存块的步长设置成 16 字节，甚至是 8 字节。

这些都是 slab 的典型应用，让我们一起来看一下这段代码：

代码 6-15

```
#define KMALLOC_BIAS_SHIFT              (5)
#define KMALLOC_MAX_SIZE            (4096)
#define KMALLOC_MINIMAL_SIZE_BIAS\
                                (1<<(KMALLOC_BIAS_SHIFT))
#define KMALLOC_CACHE_SIZE    \
            (KMALLOC_MAX_SIZE/KMALLOC_MINIMAL_SIZE_BIAS)
struct kmem_cache kmalloc_cache[KMALLOC_CACHE_SIZE]=\
                                    {{0,0,0,0,NULL,NULL,NULL},};
#define kmalloc_cache_size_to_index(size)\
    ((((size))>>(KMALLOC_BIAS_SHIFT)))

init kmalloc_init(void){
    int i;
    for(i=0;i<KMALLOC_CACHE_SIZE;i++){
    if(kmem_cache_create(&kmalloc_cache[i],\
            (i+1)*KMALLOC_MINIMAL_SIZE_BIAS,0)==NULL)
                    return -1;
    }
    return 0;
}

void *kmalloc(unsigned int size){
    int index=kmalloc_cache_size_to_index(size);
    if(index>=KMALLOC_CACHE_SIZE)
       return NULL;
    return kmem_cache_alloc(&kmalloc_cache[index],0);
}

void kfree(void *addr){
    struct page *pg;
    pg=virt_to_page((unsigned int)addr);
```

```
    kmem_cache_free(pg->cachep,addr);
}
```

首先在代码 6-15 中，我们定义了一个全局的 kmem_cache 类型结构体数组，名为 kmalloc_cache。该数组的大小为 KMALLOC_CACHE_SIZE。当然根据不同的应用情况，这个值是可以调整的。

在其他模块使用 kmalloc 函数分配内存之前，必须首先调用 kmalloc_init 函数将 kmalloc_chache 数组中每一个成员都初始化。从程序中我们可以看出，kmalloc_init 函数只不过是循环调用了 kmem_cache_create 函数而已。

在初始化完成之后，就可以使用 kmalloc 函数了，而 kmalloc 的参数只有一个，那就是想要获取的内存的大小。kmalloc 函数无非是调用 kmem_cache_alloc 从相应的 kmem_cache 结构体中分配内存而已。当然在调用 kmem_cache_alloc 函数之前，我们需要首先得到一个 kmalloc_cache 数组的索引。这很简单，只需要调用 kmalloc_cache_size_to_index 这个宏就可以了。

当 kmalloc 申请的内存不再使用后，kfree 函数就可以将它释放。kfree 函数的实现就更加简单了，首先从虚拟地址得到与之对应的 struct page 结构体，这样就可以通过 struct page 结构体的 cachep 成员得到所属的 kmem_cache 结构，然后以这个结构体作为参数调用 kmem_cache_free 释放内存。

最后，我们来运行一下 slab 和 kmalloc 的代码。

首先我们需要将代码 6-10 之后的全部代码保存到文件“mem.c”中。然后为了验证这段代码的正确性，我们需要修改一下“boot.c”的内容。

代码 6-16

```
void *kmalloc(unsigned int size);
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    test_printk();
//  timer_init();
    init_page_map();
```

```
    kmalloc_init();
    char *p1,*p2,*p3,*p4;
    p1=kmalloc(127);
    printk("the first alloced address is %x\n",p1);
    p2=kmalloc(124);
    printk("the second alloced address is %x\n",p2);
    kfree(p1);
    kfree(p2);
    p3=kmalloc(119);
    printk("the third alloced address is %x\n",p3);
    p4=kmalloc(512);
    printk("the forth alloced address is %x\n",p4);
    while(1);
}
```

这里可以随意选取几个值，使用 kmalloc 来分配内存空间并调用 kfree 将某些空间释放。读者可以比较一下最终的运行结果，根据各函数的运行原理查看一下是否正确。另外为了避免编译警告，代码 6-16 中也对 kmalloc 函数进行了声明。

最终的运行结果如下所示：

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
testing printk
test string :::   this is %s test
test char ::: H
test digit ::: -256
test X ::: 0xffffff00
test unsigned ::: 4294967040
test zero ::: 0
the first alloced address is 0x306f3000
the second alloced address is 0x306f3080
the third alloced address is 0x306f3080
the forth alloced address is 0x301e0000
```

6.4　总结

一个平台下的内存容量越大、数据访问速度越快，那么这台计算机就越“聪明”。但是很多时候我们不能向计算机提出要求，就好像一个人天生愚钝，但是却没有能力改变这种既成事实，于是唯一的希望就寄托在后天的努力上。可想而知，操作系统对于内存的处理方法就成为了计算机改变自己命运的唯一手段。从这个角度看，无论我们对内存有多么重视，都是不过分的。

再回到技术细节中，纵观本章的内容，我们实现了不同层次，不同应用的几种内存管理方法。图 6-10 总结了这些内存管理算法各层次之间的关系。

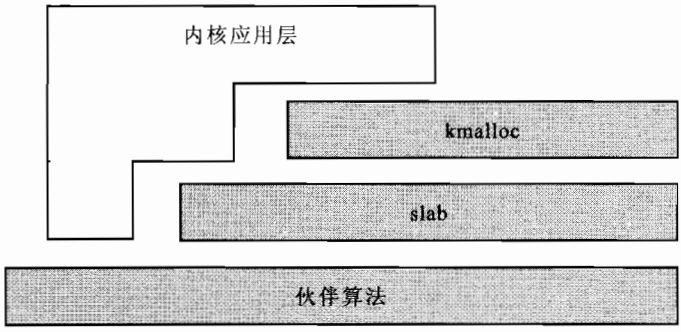


图 6-10　操作系统中的内存层次结构

如图 6-10 中所示的那样，kmalloc 函数是建立在 slab 的基础上的，slab 又是以伙伴算法为基础的，从而构成了一个全方位、立体的内存管理机制。而对于存在于操作系统内核中的其他程序，这些机制也都是公开的，所以，程序可以根据实际情况选择合适的方法获取内存，从而大大增强了内核程序的灵活性和适应能力。

当然在本章中，我们只是给出了一套自己的实现方法。程序是人写的，代码的实现方法也并不唯一。只要背后的算法原理没有变化，不同程序的实现方面没有太多优劣之分。因此，本章的代码也仅仅代表了一种参考方法，能够为读者抛砖引玉足矣。